

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Conceptos</b>	<b>3</b>
2.1. Clases y objetos . . . . .	3
2.2. Constructores . . . . .	4
2.3. Encapsulacion, Getters y Setters . . . . .	6
2.4. Enum . . . . .	9
2.5. Colecciones . . . . .	10
2.6. Widgets . . . . .	10
2.7. Polimorfismo . . . . .	11
2.8. Null Safety . . . . .	12
<b>3. Desarrollo</b>	<b>14</b>
<b>4. Resultados</b>	<b>15</b>
<b>5. Conclusión</b>	<b>16</b>

## 1. Introducción

El desarrollo de aplicaciones móviles se ha convertido en una de las áreas más dinámicas y relevantes dentro de la ingeniería de software. A lo largo del curso se han revisado conceptos fundamentales de programación, diseño de interfaces y lógica de sistemas, utilizando Flutter como marco de trabajo. Como proyecto integrador, se plantea la creación de una aplicación que simule el sistema de batallas del videojuego Pokémon, con el fin de aplicar de manera práctica los conocimientos adquiridos y demostrar la capacidad de diseñar un sistema interactivo, modular y funcional.

### 1.1. Planteamiento del problema

Los videojuegos de rol y estrategia, como Pokémon, se caracterizan por sistemas de combate basados en turnos, donde las decisiones del jugador y las características de los personajes determinan el resultado de la batalla. Replicar este tipo de mecánicas en una aplicación móvil implica enfrentar diversos retos:

- Implementar un sistema de turnos que respete las reglas de prioridad y finalización de la batalla.
- Representar las ventajas y desventajas de los tipos de ataque, siguiendo la lógica de daño doble, mitad o nulo.
- Diseñar un menú interactivo que muestre al Pokémon del usuario, al rival y las opciones de ataque disponibles.

### 1.2. Motivación

La motivación principal de este proyecto es aplicar de manera práctica los conceptos aprendidos en el curso y demostrar cómo Flutter puede ser utilizado para desarrollar aplicaciones interactivas más allá de las tradicionales interfaces de usuario. Recrear un sistema de batallas Pokémon no solo resulta atractivo por su popularidad y valor lúdico, sino que también representa un desafío técnico que fomenta:

- El pensamiento lógico y la correcta estructuración de algoritmos.
- La modularidad y reutilización de código, al definir clases y métodos que representen Pokémon, ataques y estados.
- El trabajo colaborativo, ya que cada miembro del equipo desarrolla una parte específica del proyecto (objetivos, diagramas UML, desarrollo paso a paso, resultados).
- La creatividad y motivación personal, al integrar elementos adicionales como música y estados alterados que enriquecen la simulación.

### 1.3. Objetivos

- Implementar atributos básicos de los Pokémon: vida (HP) y velocidad.
- Diseñar un sistema de turnos que determine el orden de ataque según la velocidad.
- Definir al menos un Pokémon y un ataque de cada tipo, respetando las reglas de efectividad (doble, mitad, nulo).
- Construir un menú interactivo que muestre los Pokémon en combate y las opciones de ataque.

## 2. Conceptos

En este apartado se explicaran los conceptos o las herramientas que fueron utilizadas y como funcionan.

### 2.1. Clases y objetos

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixins. Cada objeto es una instancia de una clase, y todas las clases, excepto Null, descienden de Object. La herencia basada en mixins significa que, aunque cada clase (excepto la clase superior, ¿Object?) tiene exactamente una superclase, el cuerpo de una clase se puede reutilizar en múltiples jerarquías de clases. Los métodos de extensión son una forma de añadir funcionalidad a una clase sin cambiarla ni crear una subclase. Los modificadores de clase permiten controlar cómo las bibliotecas pueden subtipificar una clase.[?]

En la programación orientada a objetos , un objeto es una unidad autocontenida de código y datos. Los objetos se crean a partir de plantillas llamadas clases. Un objeto se compone de propiedades (variables) y métodos (funciones). Un objeto es una instancia de una clase.[?]

- Utilizando miembros de clase:

Los objetos tienen miembros que consisten en funciones y datos ( métodos y variables de instancia , respectivamente). Al llamar a un método, se invoca sobre un objeto: el método tiene acceso a las funciones y datos de ese objeto. Utilice un punto (.) para referirse a una variable de instancia o método:

```
1 var p = Point(2, 2);
2
3 // Get the value of y.
4 assert(p.y == 2);
5
6 // Invoke distanceTo() on p.
7 double distance = p.distanceTo(Point(4, 4));
```

Utilice ?. en su lugar . para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
1 // If p is non-null, set a variable equal to its y value.
2 var a = p?.y;
```

- Utilizando constructores Puedes crear un objeto usando un constructor . Los nombres de los constructores pueden ser 'init' ClassName o 'init' . Por ejemplo, el siguiente código crea objetos usando los constructores 'init' y 'init': ClassName.identifierPointPoint()Point.fromJson().[?]

```
1 var p1 = Point(2, 2);
2 var p2 = Point.fromJson({'x': 1, 'y': 2});
```

El siguiente código tiene el mismo efecto, pero utiliza la newpalabra clave opcional antes del nombre del constructor:

```
1 var p1 = new Point(2, 2);
2 var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

- Variables y métodos de clase

utilice la static palabra clave para implementar variables y métodos de toda la clase.

```

1 class Queue {
2     static const initialCapacity = 16;
3
4 }
5
6 void main() {
7     assert(Queue.initialCapacity == 16);
8 }

```

- métodos estáticos

Los métodos estáticos (métodos de clase) no operan sobre una instancia y, por lo tanto, no tienen acceso a ella `this`. Sin embargo, sí tienen acceso a las variables estáticas. Como muestra el siguiente ejemplo, los métodos estáticos se invocan directamente en una clase.[?]

```

1 import 'dart:math';
2
3 class Point {
4     double x, y;
5     Point(this.x, this.y);
6
7     static double distanceBetween(Point a, Point b) {
8         var dx = a.x - b.x;
9         var dy = a.y - b.y;
10        return sqrt(dx * dx + dy * dy);
11    }
12 }
13
14 void main() {
15     var a = Point(2, 2);
16     var b = Point(4, 4);
17     var distance = Point.distanceBetween(a, b);
18     assert(2.8 < distance && distance < 2.9);
19     print(distance);
20 }

```

## 2.2. Constructores

Los constructores son funciones especiales que crean instancias de clases.

Dart implementa muchos tipos de constructores. Excepto los constructores por defecto, estas funciones utilizan el mismo nombre que su clase.[?]

1. constructores generativos Para instanciar una clase, utilice un constructor generativo.

```

1 class Point {
2     // Instance variables to hold the coordinates of the point.
3     double x;

```

```

4   double y;
5
6   // Generative constructor with initializing formal parameters:
7   Point(this.x, this.y);
8 }

```

## 2. Constructores por defecto

Si no declaras un constructor, Dart utiliza el constructor predeterminado. El constructor predeterminado es un constructor generativo sin argumentos ni nombre.<sup>[?]</sup>

## 3. Constructores con nombre

Utilice un constructor con nombre para implementar múltiples constructores para una clase o para proporcionar mayor claridad:

```

1  vconst double xOrigin = 0;
2  const double yOrigin = 0;
3
4  class Point {
5      final double x;
6      final double y;
7
8      // Sets the x and y instance variables
9      // before the constructor body runs.
10     Point(this.x, this.y);
11
12     // Named constructor
13     Point.origin() : x = xOrigin, y = yOrigin;
14 }

```

Una subclase no hereda el constructor con nombre de la superclase. Para crear una subclase con un constructor con nombre definido en la superclase, implemente dicho constructor en la subclase.

## 4. Constructores constantes

Si tu clase produce objetos que no cambian, haz que estos objetos sean constantes en tiempo de compilación. Para ello, define un `const` constructor con todas las variables de instancia establecidas como `final`.<sup>[?]</sup>

```

1  class ImmutablePoint {
2      static const ImmutablePoint origin = ImmutablePoint(0, 0);
3
4      final double x, y;
5
6      const ImmutablePoint(this.x, this.y);
7  }

```

Los constructores constantes no siempre crean constantes. Pueden invocarse fuera de un `const` contexto. Para obtener más información, consulte la sección sobre el uso de constructores

## 5. Constructores de redireccionamiento

Un constructor puede redirigir a otro constructor de la misma clase. Un constructor que redirige tiene un cuerpo vacío. El constructor utiliza `this` en lugar del nombre de la clase después de dos puntos (`:`).

```
1 class Point {
2     double x, y;
3
4     // The main constructor for this class.
5     Point(this.x, this.y);
6
7     // Delegates to the main constructor.
8     Point.alongXAxis(double x) : this(x, 0);
9 }
```

## 2.3. Encapsulacion, Getters y Setters

En Dart, la encapsulación consiste en ocultar datos dentro de una biblioteca, protegiéndolos de factores externos. Esto ayuda a controlar el programa y evita que se vuelva demasiado complejo.[?]

La encapsulación se puede lograr mediante:

- Declarar las propiedades de la clase como privadas utilizando guion bajo (`_`).
- Proporcionar métodos públicos `getter` y `setter` para acceder y actualizar el valor de la propiedad privada.

Nota: Dart no admite palabras clave como `'public'`, `'private'` y `'protected'`. Dart utiliza “`_`” (guion bajo) para indicar que una propiedad o método es privado. Los métodos `getter` y `setter` se utilizan para acceder y actualizar el valor de una propiedad privada. Los métodos `getter` se utilizan para acceder al valor de una propiedad privada. Los métodos `setter` se utilizan para actualizar el valor de una propiedad privada.[?]

Ejemplo 1: En este ejemplo, crearemos una clase llamada **Employee**. Esta clase tendrá dos propiedades privadas: `_id` y `_name`. También crearemos dos métodos públicos, `getId()` y `getName()`, para acceder a las propiedades privadas. Asimismo, crearemos dos métodos públicos, `setId()` y `setName()`, para actualizar las propiedades privadas.[?]

```
1 class Employee {
2     // Private properties
3     int? _id;
4     String? _name;
5
6     // Getter method to access private property _id
7     int getId() {
8         return _id!;
9     }
10    // Getter method to access private property _name
11    String getName() {
12        return _name!;
13    }
14    // Setter method to update private property _id
```

```

15     void setId(int id) {
16         this._id = id;
17     }
18     // Setter method to update private property _name
19     void setName(String name) {
20         this._name = name;
21     }
22
23 }
24
25 void main() {
26     // Create an object of Employee class
27     Employee emp = new Employee();
28     // setting values to the object using setter
29     emp.setId(1);
30     emp.setName("John");
31
32     // Retrieve the values of the object using getter
33     print("Id: ${emp.getId()}");
34     print("Name: ${emp.getName()}");
35 }

```

En Dart, puedes controlar el acceso a las propiedades e implementar la encapsulación mediante el uso de propiedades de solo lectura. Para ello, añade la palabra clave ‘final’ antes de la declaración de la propiedad. De esta forma, solo podrás acceder a su valor, pero no modificarlo.[?]

```

1 class Student {
2     final _schoolname = "ABC School";
3
4     String getSchoolName() {
5         return _schoolname;
6     }
7 }
8
9 void main() {
10     var student = Student();
11     print(student.getSchoolName());
12     // This is not possible
13     //student._schoolname = "XYZ School";
14 }

```

Puedes crear métodos getter y setter usando las palabras clave **get** y **set**. En el siguiente ejemplo, hemos creado una clase llamada **Vehicle**. Esta clase tiene dos propiedades privadas: **\_model** y **\_year**. También hemos creado dos métodos getter y setter para cada propiedad. Estos métodos se llaman **model** y **year**, y se utilizan para acceder y actualizar el valor de las propiedades privadas.[?]

```

1 class Vehicle {
2     String _model;
3     int _year;

```

```

4
5 // Getter method
6 String get model => _model;
7
8 // Setter method
9 set model(String model) => _model = model;
10
11 // Getter method
12 int get year => _year;
13
14 // Setter method
15 set year(int year) => _year = year;
16 }
17
18 void main() {
19     var vehicle = Vehicle();
20     vehicle.model = "Toyota";
21     vehicle.year = 2019;
22     print(vehicle.model);
23     print(vehicle.year);
24 }

```

¿Por qué es importante la encapsulación?

1. Ocultación de datos : La encapsulación oculta los datos del exterior. Impide que el código fuera de la clase acceda a ellos. Esto se conoce como ocultación de datos.[?]
2. Facilidad de prueba : La encapsulación permite probar la clase de forma aislada. Esto permite probar la clase sin necesidad de probar el código fuera de ella.[?]
3. Flexibilidad : La encapsulación permite cambiar la implementación de la clase sin afectar al código que se encuentra fuera de la clase.[?]
4. Seguridad : La encapsulación permite restringir el acceso a los miembros de la clase. Esto permite limitar el acceso a los miembros de la clase desde el código externo a la biblioteca.[?]

con todo lo anterior la aplicacion de estos conceptos para el proyecto en cuestion se observa de la siguiente manera:

```

1 class Pokemon {
2     String nombre;
3     TipoPokemon tipo;
4     int hp;
5     int velocidad;
6     List<Ataque> ataques;
7
8     String imagePath;
9
10 }

```

## 2.4. Enum

En Dart, los enums extienden de la clase Enum, y nos sirven para agrupar conjuntos de valores como veremos más adelante.[?]

para declarar una enumeración simple, debemos usar la palabra clave enum el nombre debe empezar con mayúscula y allí seguido una lista de valores a enumerar.[?]

```
1 enum Color { red , green , blue }
2 enum DayOfWeek {
3     monday ,
4     tuesday ,
5     wednesday ,
6     thursday ,
7     friday ,
8     saturday ,
9     sunday
10 }
```

Dart también permite en las clases de enums declarar atributos, funciones y constructores.

Para declarar enums mejorados, se usa de la misma forma que las clases normales, con algunas particularidades:

- las variables o atributos deben ser final.
- Todos los constructores deben ser constantes.
- Enum se extiende automáticamente.
- No se pueden crear variables con el nombre values porque entra en conflicto con la propiedad de los enums.
- Todos los casos de enums deben ser declarados al principio.

en el proyecto:

```
1 enum TipoPokemon {
2     FUEGO,
3     AGUA,
4     PLANTA,
5     ELECTRICO,
6     ROCA,
7     NORMAL,
8     LUCHA,
9     VENENO,
10    TIERRA,
11    VOLADOR,
12    PSIQUICO,
13    BICHO,
14    FANTASMA,
15    DRAGON,
16    DARK,
17    STEEL
18 }
```

```

1 var lanzallamas =
2   Ataque(nombre: 'Lanzallamas', tipo: TipoPokemon.FUEGO, poder: 90);
3
4 var placaje =
5   Ataque(nombre: 'Placaje', tipo: TipoPokemon.NORMAL, poder: 40);
6 var pistolaAgua =
7   Ataque(nombre: 'Pistola Agua', tipo: TipoPokemon.AGUA, poder: 40);

```

## 2.5. Colecciones

En un mapa, cada elemento es un par clave-valor. Cada clave dentro de un par está asociada a un valor, y tanto las claves como los valores pueden ser de cualquier tipo de objeto. Cada clave solo puede aparecer una vez, aunque el mismo valor puede estar asociado a varias claves diferentes. La compatibilidad de Dart con los mapas se proporciona mediante literales de mapa y el Map tipo.[?]

para el proyecto:

```

1 const Map<TipoPokemon, Map<TipoPokemon, double>> mapaDeTipos =

```

Quizás la colección más común en casi todos los lenguajes de programación sea el array, o grupo ordenado de objetos. En Dart, los arrays son Listobjetos, por lo que la mayoría de la gente los llama simplemente listas.[?]

Las literales de lista de Dart se representan mediante una lista de elementos separados por comas y encerrados entre corchetes ( []). Cada elemento suele ser una expresión. A continuación, se muestra una lista de Dart sencilla:

Las listas utilizan indexación basada en cero, donde 0 es el índice del primer elemento y 0 list.length - 1 es el índice del último. Puede obtener la longitud de una lista mediante la .lengthpropiedad y acceder a sus elementos mediante el operador de subíndice ( []):

```

1 List<String> battleLog = [ '¡La batalla ha comenzado!' ];

```

## 2.6. Widgets

La Interfaz de Usuario en Flutter, está compuesta de Widgets. Los Widgets en Flutter se construyen utilizando un Framework moderno inspirado en React. Casi todo en Flutter está compuesto de Widgets. Los Widgets describen como sería su vista dado una configuración y estado actual. Cuando el estado cambia, el Widget se reconstruye.[?]

Los Widgets son subclases de StatelessWidget o StatefulWidget. Todo depende de si se quiere guardar el estado o no del widget. Todos los widgets deben implementar la función build(), dentro de este se va armando la estructura necesaria para el widget requerido. El StatelessWidget se utiliza cuando no es necesario conservar el estado del widget, para guardar el estado o alguna otra característica de un widget es recomendable usar StatefulWidget o algún manejador de estados.[?]

Widgets básicos

- Text: Es un widget que permite crear textos con estilos personalizados.
- Row: Es un widget que permite tener un diseño flexible y tener widgets hijos en un conjunto horizontal (Fila).

- Column: Es un widget al igual que el Row, es de diseño flexible y permite tener widgets hijos en un conjunto vertical (Columna).
- Stack: El widget Stack permite colocar widgets uno encima de otro. Es un widget que posiciona a sus hijos en relación con los bordes de su caja. Aquí se puede utilizar otro widget Positioned que se encarga de controlar en dónde se van a ubicar los hijos de Stack.
- Container: El widget Container permite crear un elemento rectangular, el cual se puede decorar con estilos, el fondo, sombras y formas con otro widget BoxDecoration. El widget BoxDecoration básicamente es un widget que nos permite pintar o dibujar una caja en blanco a nuestra necesidad.

En el proyecto

```

1  return Scaffold(
2      appBar: AppBar(
3          title: const Text('Batalla_Pokémon'),
4      ),

1  Widget _buildAttackMenu() {
2
3      //Comprueba si la batalla termina
4      bool batallaTerminada = (pokemonUsuario.hp == 0 || pokemonRival.hp == 0);
5      //Obtiene los ataques de nuestro Pokémon
6      List<Ataque> ataques = pokemonUsuario.ataques;
7
8      //Crea una lista de Widgets (botones)
9      List<Widget> botonesDeAtaque = [];
10
11     for (var ataque in ataques) {
12         botonesDeAtaque.add(
13             Padding(
14                 padding: const EdgeInsets.symmetric(vertical: 4.0),
15                 child: ElevatedButton(
16                     //Si la batalla terminó, onPressed es null
17                     onPressed: batallaTerminada
18                         ? null
19                         : () {
20                             _atacar(ataque);
21                         },
22                     child: Text('${ataque.nombre}_(${ataque.tipo.name}')),
23                 ),
24             ),
25         );
26     }

```

## 2.7. Polimorfismo

Poli significa muchos y morfo significa formas. El polimorfismo es la capacidad de un objeto para adoptar muchas formas. Como seres humanos, tenemos la capacidad de adoptar muchas formas. Podemos ser estudiantes, profesores, padres, amigos, etc. De manera similar, en la programación orientada a objetos, el polimorfismo es la capacidad de un objeto para adoptar muchas formas.[?]

La sobrescritura de métodos es una técnica que permite crear un método en la clase hija con el mismo nombre que el método en la clase padre. El método en la clase hija sobrescribe el método en la clase padre.[?]

```
1 class ParentClass{
2 void functionName(){
3 }
4 }
5 class ChildClass extends ParentClass{
6 @override
7 void functionName(){
8 }
9 }
```

Ejemplo 1: Polimorfismo mediante la sobrescritura de métodos en Dart

En el siguiente ejemplo, existe una clase llamada Animal con un método llamado eat() . El método eat() se redefine en la clase hija llamada Dog .

```
1 class Animal {
2 void eat() {
3 print("Animal_is_eating");
4 }
5 }
6
7 class Dog extends Animal {
8 @override
9 void eat() {
10 print("Dog_is_eating");
11 }
12 }
13
14 void main() {
15 Animal animal = Animal();
16 animal.eat();
17
18 Dog dog = Dog();
19 dog.eat();
20 }
```

Ventaja de usar polimorfismo:

1. Las subclases pueden anular el comportamiento de la clase padre.
2. Nos permite escribir código más flexible y reutilizable.

## 2.8. Null Safety

La seguridad nula evita errores que resultan del acceso no intencional a las variables establecidas en null. Por ejemplo, si un método espera un entero pero recibe null, la aplicación genera un error de ejecución. Este tipo de error, un error de desreferencia nula, puede ser difícil de depurar.

Con seguridad nula sólida, todas las variables requieren un valor. Esto significa que Dart considera que todas las variables no aceptan valores nulos. Solo se pueden asignar valores del tipo declarado, como `int i=42`. Nunca se puede asignar un valor de nulo los tipos de variable predeterminados. Para especificar que un tipo de variable puede tener un nullvalor, agregue un `?` después de la anotación de tipo: `int? i`. Estos tipos específicos pueden contener un valor de null o un valor del tipo definido.

La seguridad de nulos convierte los posibles errores de ejecución en errores de análisis en tiempo de edición. Con la seguridad de nulos, el analizador y los compiladores de Dart detectan si una variable no nula presenta alguna de las siguientes características:

No se ha inicializado con un valor distinto de nulo Se le ha asignado un nullvalor.

Dart admite la seguridad nula utilizando los siguientes dos principios de diseño básicos:

- No nulo por defecto: A menos que se indique explícitamente a Dart que una variable puede ser nula, se considera no nula. Esta opción predeterminada se eligió después de que una investigación revelara que la opción no nula era, con diferencia, la más común en las API.
- Totalmente sano: La seguridad nula de Dart es sólida. Si el sistema de tipos determina que una variable o expresión tiene un tipo no nulo, se garantiza que nunca podrá evaluarse nullo en tiempo de ejecución.

En el código:

```
1 late Pokemon pokemonUsuario;  
2 late Pokemon pokemonRival;  
3 late Batalla batalla;
```

Con setters que aceptan solo valores no nulos:

```
1 pokemonUsuario = Pokemon(  
2     nombre: 'Charizard',  
3     tipo: TipoPokemon.FUEGO,  
4     hp: 78,  
5     velocidad: 100,  
6     ataques: [lanzallamas, placaje],  
7     //ruta de pubspec.yaml  
8     imagePath: 'assets/imagenes/charizard.png',  
9 );
```

### **3. Desarrollo**

Segunda prueba

## 4. Resultados

probando... un dos tres un dos tres

## **5. Conclusión**